Improving Testing of Multi-Unit Computer Players for Unwanted Behavior using Coordination Macros

Malik Atalla and Jörg Denzinger

Abstract— **We present an improvement to behavior testing of computer players based on evolutionary learning of cooperative behavior that extends the known approach to allow for so-called coordination macros. These macros represent knowledge about the application and are interpreted by the agents that are testing the computer player based on the current situation to achieve coordination between the agents. Our experimental evaluation using this approach to test computer players for one competition scenario of the ORTS real-time strategy game showed that the macros enabled the testing system to find weaknesses much faster than the previous approach, respectively to find weaknesses that the previous approach was not able to find within the given resource limit.**

I. Introduction

Testing of complex software systems is a very difficult task. While testing properties of a system that can be described positively in form of use cases is mostly difficult due to the potentially big number of variants of such use cases, there are usually also quite a number of properties that are described "negatively", i.e. as behaviors that the system should *not* show or situations that should *not* be reached by any behavior that the system is showing. Testing for such unwanted behavior (often also called emergent misbehavior, see [1]) requires from a tester to figure out interactions with the tested system that lead to showing the unwanted situation or behavior and if a tester is not able to come up with any sequence of interactions that creates unwanted behavior within certain resource limits, then the assumption is that it is not likely that the system will show the unwanted behavior.

But for many complex systems, we find that there are groups of users that are more interested in creating unwanted behavior than in using the system in the intended manner, which naturally makes it more likely that unwanted behavior will be found. It has become a kind of sport for some groups of users to find so-called sweet spots of commercial computer games, i.e. ways of achieving some goal in the game that the game designers did not intend to be possible. And the Internet provides an easy way to let others know about such game weaknesses (see, for example, [2]). Also, in game-like competitions of more or less intelligent computer programs, like TAC (trading agent competition, see [3]) or the ORTS competitions (**O**pen **R**eal-**T**ime **S**trategy Game, see [4]), finding weaknesses in another competitor (before the designer of the competitor does) can provide the deciding advantage against this opponent in the next competition. And this puts previous game competitors under a lot of scrutiny, again. In both cases, good testing for unwanted behavior should reduce the likelihood for such problems to materialize.

While testing for unwanted behavior is still mostly the domain of human experts with nearly non-existing tool support, there has been some work that uses *learning of behavior* to at least provide assistance with this task (see, for example, [5], [6]). These works provide starting points for better testing for unwanted behavior, but a lot of additional work is needed to provide human testers with adequate support in their tasks. A major research topic within this area is how to integrate general and application-specific knowledge into the learning procedure to allow for longer and/or more complex interactions with the system to be tested.

In this paper, we present an improvement of the method from [5] that allows for the use of so-called *coordination macros* that provide knowledge about good action combinations for groups of agents. The general idea is to use evolutionary learning of action sequences that create the behaviors of agents that interact with the system that is tested. By measuring how near the tested system comes to an unwanted behavior, we have a fitness measure that is used to drive the evolution towards action sequences that finally produce in the tested system the unwanted behavior. The agents interacting with the tested system in [5] were extremely simple, essentially just executing exactly the next action in their action sequences.

We extend these agents to interpret the given action based on the current situation they observe. This allows us not only to have some kind of conditional actions in action sequences, it also provides us with a way to have actions spanning all or a group of the agents in a coordinated manner for some time to achieve a particular result or perform a difficult combined action. We evaluated this extension by testing competition systems for one of the scenarios of the ORTS competition to reveal weaknesses that other competition systems might exploid. Our evaluation showed that our extension finds weaknesses much faster than the original approach of [5], if the original approach is able to find any weaknesses, and the extension also finds weaknesses in players for which the approach from [5] was not able to find weaknesses (within the resource limits).

II. Our general testing approach for unwanted behavior

In this section, we will first present the general idea of behavior testing using learning and then we will present the

M. Atalla is with the TU Berlin (email: malikatalla@hotmail.com) and J. Denzinger is with the University of Calgary (email: denzinge@cpsc.ucalgary.ca).

978-1-4244-4815-9/09/$25.00 ©2009 IEEE

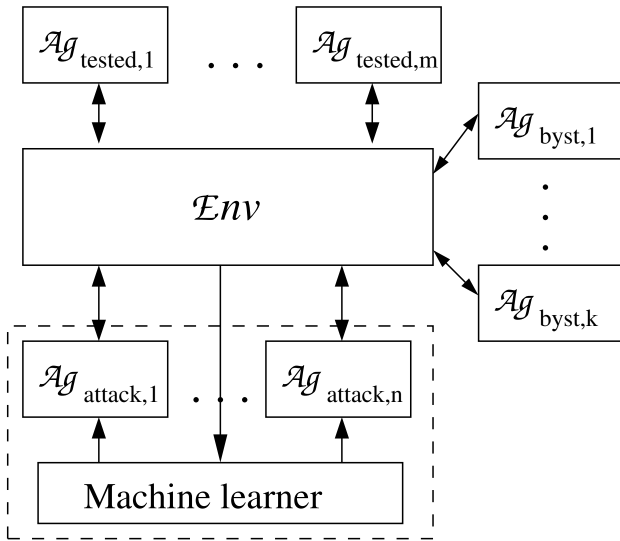

Fig. 1. General setting for testing for unwanted behavior

instantiation of this general idea as given in [5] and extended to multi-agent systems (MAS) in [7].

A. Testing for unwanted behavior using learning

From the view of a tester, the system to be tested can be seen as a multi-agent system A_{tested} = $\{\mathcal{A}g_{tested,1},...,\mathcal{A}g_{tested,m}\}$ that acts in an environment $\mathcal{E}nv$. In the same environment, there are also other agents, representing users of A_{tested}, other systems A_{tested} interacts with or a part of the system to be tested that is not tested. These other agents can be divided into the set A_{attack} = $\{\mathcal{A}g_{attack,1},...,\mathcal{A}g_{attack,n}\}$ of agents that the tester can control and use to "attack" A_{tested}[1] and the set A_{byst} = $\{\mathcal{A}g_{byst,1},...,\mathcal{A}g_{byst,k}\}$ that contains all the other agents that the tester cannot control or has chosen not to control and that are not part of the system or system part that is tested.

The idea of testing using learning of behavior is as indicated by Figure 1. A machine learner directs the agents in A_{attack} by creating for each $\mathcal{A}g_{attack,i}$ a strategy that this agent uses to interact with the agents of A_{tested} and A_{byst} in $\mathcal{E}nv$ (or a simulation of it). The machine learner then analyses the resulting behavior of the tested agents and everything else that happened in $\mathcal{E}nv$ and creates new strategies for the attack agents. This cycle is repeated until either a resource limit is reached or the agents in A_{tested} showed an unwanted behavior.

In the following we will describe the set of actions of an agent $\mathcal{A}g_{attack,i}$ by the set $Act_{attack,i}$ and then a particular set of strategies for the attack agents can be described as a sequence of actions for each agent in A_{attack}:

$((a_{1,1},...,a_{1,l}),...,(a_{n,1},...,a_{n,l}))$,

with $a_{i,j} \in Act_{attack,i}$. These action sequences, executed by the appropriate agents in the environment, then produce a trace e_0,e_1,...,e_l of environmental states[2]. Note that due to different behavior of bystander agents two evaluations of a set of strategies for the attack agents might not lead to the same traces.

B. Testing using evolutionary learning of behavior

In theory, every learner that can produce action sequences for agents could be used as the learner of the last subsection. In [7], an evolutionary learner was suggested that has as general form of an individual the vector of action sequences $((a_{1,1},...,a_{1,l}),...,(a_{n,1},...,a_{n,l}))$ that we used to describe the interactions of the agents of $\mathcal{A}g_{attack,i}$ with the other agents and the environment.

An individual's fitness is evaluated by having each of the agents in A_{attack} execute the actions of its sequence within the environment in the given sequence and having the agents of A_{tested} and A_{byst} react to the actions of the attack agents. In [7], each $a_{i,j}$ corresponds directly to an element of the set $Act_{attack,i}$ of the agent $\mathcal{A}g_{attack,i}$. In [5], this was also true, but the particular action was made dependent on the current environmental situation the agent was in, by having the action represented as a number and for each situation the action associated with a number could be different (making sure that one of the actions possible in the particular situation would be performed since the agent would not try to perform any actions that are not possible in the situation, so essentially the agent performed action $a_{i,j}(e_j)$). The concrete fitness function evaluating an individual would then evaluate the sequence of environmental states e_0,e_1,...,e_l that the individual produced. The general scheme for the fitness fit is

$$fit((e_0,...,e_x)) = \begin{cases} \text{j, if } \mathcal{G}_{test}((e_0,...,e_j)) = \text{true and} \\ \quad \mathcal{G}_{test}((e_0,...,e_i)) = \text{false for all } i < j \\ \sum_{i=1}^{x} near_goal((e_0,...,e_i)), \text{ otherwise.} \end{cases}$$

In this scheme, $\mathcal{G}_{test}$ is a predicate that is true, if the sequence of environmental states it has as argument shows the unwanted behavior we are testing for. And $near_goal$ is intended to measure how near its argument trace comes to fulfilling $\mathcal{G}_{test}$ and the smaller its value is, the nearer the trace came to the goal. Naturally, both depend on the particular system and application that is tested.

New individuals are created using the usual genetic operators for sequences, but these operators can work on different levels (and there are usually also several variants to each operator that come to mind). The general fitness scheme above allows for a special kind of genetic operators that we call *targeted operators* (see [6]). Since fit measures the nearness to producing the behavior we test for after every environmental state, it is possible to identify actions in the action sequences that lead the attack team away from the test goal. Targeted operators create new individuals the same

[1] Since "tested" and "tester" can be easily confused we have chosen to call these agents attack agents instead of tester agents.

[2] We are assuming here a synchronous system, i.e. each agent in A_{attack} executes the i-th action in its sequence at the same time and the combination of these actions then creates e_i. This is what is required in our application. But by introducing a number of time units after each action in each sequence, we can also explicitly model the time between actions.

way as standard operators, but instead of choosing random positions or agents they target positions that are a little bit before actions that the fitness function identifies as "bad" with regard to coming nearer to the goal.

With the above pieces, the learner can use a standard evolutionary algorithm to perform the learning: it creates an initial population by creating random individuals. It evaluates these individuals for their fitness and uses one of the known selection schemes that combine fitness with some random factors to select individuals to apply the genetic operators on. It creates a certain number of new individuals and replaces the least fit individuals in the current population with these new individuals to form the next population and it repeats the creation of a new population until either the learning goal is fulfilled or a resource limit (in form of a number of generations or a time limit) is reached.

III. Integrating cooperation macros

[7] has shown that the basic method presented in the last section is able to reveal weaknesses in a system of cooperating agents A_{tested} that were rescuing survivors in an earthquake scenario. But if we look at the found problems, we can observe that it was not necessary for the agents in A_{attack} to cooperate with each other very much. If the testing requires coordination of the agents in A_{attack}, especially over a sequence of several actions, then the basic method requires many "lucky" applications of genetic operators and random effects to "construct" a successful attack.

A solution to this problem is to extend the actions of the attack agents by *macro actions* that are expanded by the agents during the fitness evaluation. This expansion can then take into account the current state of the environment, including previous and planned actions of the agent itself and the other attack agents (essentially introducing the equivalent of communication between the attack agents to achieve coordination). What macro actions should be introduced depends on the particular set A_{tested} of agents to be tested, so that this approach is less general than the one in [7], but this is well in line with what we see human testing experts do by using domain knowledge and general knowledge about testing.

More formally, we extend each of the sets $Act_{attack,i}$ by a set Act_{macro}, i.e. $Act^{new}_{attack,i} = Act_{attack,i} \cup Act_{macro}$, $Act_{attack,i} \cap Act_{macro} = \emptyset$, and each agent $\mathcal{A}g_{attack,i}$ itself is extended by some internal data fields Dat_i for memory and a *decision function* $f_{\mathcal{A}g_{attack,i}} : Act^{new}_{attack,1} \times ... \times Act^{new}_{attack,n} \times \mathcal{E}nv \times Dat_i \rightarrow Act_{attack,i}$ to determine what "normal" action or sequence of "normal" actions it is taking during a fitness evaluation run. Dat_i contains two variables, namely an element $a^i \in Act_{macro} \cup \{\perp\}$ that contains the macro action the agent is currently working on (respectively $\perp$ if the agent is not involved in such an action at the moment) and an element $k^i \in \mathbb{N}$ that indicates how many "steps" to do are left in the action sequence that implements the macro action in a^i.

If we evaluate an individual $((a_{1,1},...,a_{1,l}),...,(a_{n,1},...,a_{n,l}))$ (now with $a_{i,j} \in Act^{new}_{attack,i}$), this produces not only a sequence $e_0,e_1,...,e_l$ of environmental states, for each agent we also get sequences $a^i_0,a^i_1,...,a^i_l$ of executed macro actions and $k^i_0,k^i_1,...,k^i_l$ of step counts, where $a^i_0 = \perp$ and $k^i_0 = 0$. As stated above, the action done by $\mathcal{A}g_{attack,i}$ at step j is now not always $a_{i,j}$, instead it is the result of $f_{\mathcal{A}g_{attack,i}}(a_{1,j},...,a_{n,j},e_j,(a^i_j,k^i_j))$, which is defined as follows:

$$f_{\mathcal{A}g_{attack,i}}(a_{1,j},...,a_{n,j},e_j,(a^i_j,k^i_j)) =$$

$$\begin{cases} a_{ij}(e_j), & \text{if } k^i_j = 0 \text{ and } a_{sj} \notin Act_{macro} \text{ for all } s \in \{1,...,n\} \\ app(a^i_j,e_j,k^i_j,i), & \text{if } k^i_j \neq 0 \\ app(a_{sj},e_j,k_{max}(a_{sj}),i), & \text{if } a_{sj} \in Act_{macro} \text{ and } s \text{ is minimal index with } a_{sj} \in Act_{macro} \text{ and } a_{sj} \text{ requires } \mathcal{A}g_{attack,i} \text{ to participate in its execution} \end{cases}$$

Here, *app* is a function that takes a macro action, an environmental state, a number and an agent number and generates the appropriate action for the agent for this state, this macro and the number indicating which action in the sequence implementing the macro is needed. This is naturally dependent on the particular application. k_{max} is another function that returns the number of actions in the sequence implementing a macro action. And, as stated before, we have that the particular action performed by an agent is always dependent on the environment, which we indicate by adding the environmental state as argument to the action from the individual (i.e. $a_{ij}(e_j)$).

Additionally, we have that

$$a^i_{j+1} = \begin{cases} a^i_j, & \text{if the execution of the macro action is not finished yet} \\ a_{sj}, & \text{if } a_{sj} \in Act_{macro} \text{ and its execution has just started} \\ \perp, & \text{otherwise} \end{cases}$$

and

$$k^i_{j+1} = k^i_j \dot{-} 1$$

Although we want the description of the incorporation of macros in this section to be as general and inclusive as possible, we made some decisions that for some applications might not be the best. While the attack agents are executing a macro action, the actions in their strategy sequences that normally would be performed are ignored. An alternative to this would be to interrupt the sequences the moment a macro starts and then resume following them when the macro is finished. We have not done this here, since we are looking into synchronous execution of the action sequences and with this usually comes a limitation of the number of steps to do. So, it was either "overwriting" some actions or cutting at the end and we have chosen the first. Another decision that might be changed in some applications is that we perform a macro for a fixed number of actions (indicated by k_{max}). Alternatively, a condition on the environmental state could be used to decide when to stop doing the macro.

There are possibilities for special genetic operators dealing with macro actions (as can be seen in Section IV-B), but we do not see these operators as something that should be in every instantiation of our method. Therefore we do not provide here any such special operators. Obviously, there is also no need to modify the general scheme for fitness evaluation from the last section, since in the end our agents still produce a sequence of environmental states that should be evaluated as described before.

IV. INSTANTIATING OUR APPROACH TO ORTS

In the following, we will first give a brief introduction to the ORTS environment and then instantiate the testing approach from the last section to test computer players for ORTS.

A. ORTS

The ORTS project (see [4]) was initiated by Michael Buro and with the help of quite some contributers has developed into an environment that can be used by AI researchers to evaluate their ideas for AI methods that can be used in real time strategy games. Real time strategy games (RTSs) are a rather popular genre within commercial computer games (very successful games are the Age of Empires series or the Warcraft series) that has a player in control of a rather large number of units and installations creating at least some kind of economy (to support the installations and units using resources) and usually with the goal to defeat other players by destroying their installations and units. Commercial RTS come with computer players of various difficulty that offer some challenges at first, but usually human players learn rather quickly how to defeat such players. And this is just one reason why having an interface for AI researchers into such a game would offer quite some interesting possibilities, like having computer players that reason on the actions of their units, plan well ahead and are even able to learn over time (both within a game or over a sequence of games).

ORTS was developed with the explicit goal to provide researchers with such an interface. It consists of a server that "runs" a game in rounds and that allows clients to connect and assume the roles of players in the game. The clients are only responsible for determining the actions of their units to be taken in the current game round and sending them to the server. From that the server calculates the game state for the next round. View updates are determined for each client individually, so that various variants of the game with regard to information about the game state are possible. In fact, one of the major goals in the development of ORTS was to provide a lot of flexibility in the creation of various game variants.

ORTS is also used in annual competitions that pit computer players developed by various research groups against each other. In a competition, there are different types of games ranging from a full game (i.e. having the players start with few units and only base installations, explore the world, enhance their technology, fight other players until one player has defeated all other players) to a game that just represents tactical combat between groups of units for two players (which is an often occurring "sub-game" within a real time strategy game).

For evaluating our extension of the testing approach by learning of cooperative behavior, we used a slightly scaled down version of the tactical combat game that was offered as game 4 in the 2007 ORTS Tournament (we were not able to find the codes for the 2008 Tournament entries at the time of the development of our testing system). It contains only one type of unit, called marine. The actions they can take are moving towards some point in the field with some speed of their choosing and shooting at an enemy unit, if in range. There is no "fog-of-war" in this type of game, so that all the clients have complete knowledge of the game state. The scenario has been modified to contain only 10 instead of 50 marines per team and is played on a smaller map (20 by 20 instead of 64 by 48). The length of a game was limited to 200 rounds. This "simplification" at least gave the original testing method of [7] a chance to find some weaknesses in the tested players, but, as Section V shows, the simple variant is on the border of what the method of [7] can deal with.

B. Testing ORTS players

In this section, we will first present how we instantiated the basic concept from [7] for testing computer players for the ORTS game described in the previous subsection that are intended for the ORTS annual competition. Then we will describe what macro actions we created for ORTS and how we fit these macros into the basic instantiation.

1) Testing ORTS using learning of behavior: To instantiate the general setting for testing for unwanted behavior of a computer player for ORTS, the environment $\mathcal{E}nv$ is the game on the side of the ORTS server. The system A_{tested} is the computer player that we want to test, so that each $\mathcal{A}g_{tested,i}$ is one of the units that the player controls. With regard to A_{attack}, each $\mathcal{A}g_{attack,i}$ represents a unit of the opponent of the computer player, so that the machine learner is trying to learn one opponent strategy that results in unwanted behavior of the computer player. The game variant of ORTS that we are using allows for additional units, namely randomly moving, invincible sheep that obviously would be modelled in the general setting as bystander agents. But for our tests, we turned this feature of the game off (which allowed us to evaluate an individual in our evolutionary learner using just one game run, since there were no random effects).

As described in Section II-B, an individual for the evolutionary algorithm consists of an action sequence for each of the attack units. The sets $Act_{attack,i}$ were all identical and consist of the following actions:

- move (with maximum speed) right: $right$
- move (with maximum speed) left: $left$
- move (with maximum speed) up: up
- move (with maximum speed) down: $down$
- attack closest opponent unit in range: $aclose$
- attack weakest opponent unit in range: $aweak$

These actions represent a subset of the possible actions of a unit in ORTS. While ORTS allows as a move action the

movement to a particular coordinate at a particular speed (that might take several rounds and obviously can be interrupted/cancelled in subsequent rounds), the concept from [7] does not allow for parameterised actions. With regard to where to go, we see our interpretation as fulfilling the same goals (without having to deal with cancelling actions). Also, most computer players use always maximum speed, so that we do not think that we have limited ourselves (or our attack agents) in any way. Providing two attack actions dependent on the current state of the game is a limitation compared to what ORTS itself offers (by allowing to attack any particular unit in range), but we think that providing a specific attack action for each possible unit of the tested player would not have changed the success, respectively lack thereof, of the base version, since the issue really is that attacks need to be coordinated. And then the two attack actions are enough.

For the fitness evaluation of an individual, we fixed the random seed of the server so that the marines are placed at the same initial positions in every evaluation of a particular run of our test system. With the condition $\mathcal{G}_{test}$ we want to find weaknesses in the tested computer player and losing the game is obviously a weakness of the player. With regard to defining a win or loss, this is obvious if one player has lost all units, but since we run the game only for a certain number of rounds, we also need to define a winner (or a draw) if both players have still units at the end of the game. There are several possibilities for this definition and we have chosen to count the number of units remaining and declare the player with more units the winner (independent of how "healthy" the units are). If $ucount_{attack}(e_j)$ and $ucount_{tested}(e_j)$ are the numbers of units still alive in e_j from A_{attack} and A_{tested}, then we have

$\mathcal{G}_{test}((e_0,...,e_j))$ = true, if there is an $i \leq j$ such that $ucount_{tested}(e_i) = 0$ or if $j = l$ and $ucount_{attack}(e_l) > ucount_{tested}(e_l)$

This definition of $\mathcal{G}_{test}$ provides also a first idea for the definition of $near_goal$, namely to use the difference in the unit count to evaluate a strategy for the agents in A_{attack}. But preliminary tests showed that using just this criterion was not enough, since it does not provide enough difference between individuals, especially at the beginning of the evolutionary learning process, so that we implemented in our system several criteria that can be combined. These criteria are

Surviving Units: As already mentioned, the difference in unit numbers is obviously an important criterion. We created a measure SU as follows:

$$SU(e_j) = ucount_{attack}(e_j) * w_{uc,att} + ucount_{tested}(e_j) * w_{uc,test}$$

Here, $w_{uc,att}$ and $w_{uc,test}$ are weight parameters and while $w_{uc,att}$ should be positive, $w_{uc,test}$ always should be negative.

Health: The health of an individual unit is a good indicator how likely it will be that the unit can continue to fight. Our measure He sums up the health of all units of a player:

$$He(e_j) = \sum_{i=1}^{n} he(\mathcal{A}g_{attack,i}, e_j) * w_{he,att} + \sum_{i=1}^{n} he(\mathcal{A}g_{tested,i}, e_j) * w_{he,test}$$

Here, $he(\mathcal{A}g, e_j)$ denotes the number of health points of the unit/agent $\mathcal{A}g$ in the game state e_j. As in case of SU, we are using weight parameters to be able to change the impact of this measure on the overall fitness. $w_{he,att}$ as weight for the health of the attack agents should be positive, while $w_{he,test}$ should be negative.

Unengaged Units: In order to damage other units a unit needs to be in shooting range to such a unit. If a lot of the own units cannot attack enemy units then this represents a not so favourable situation, while a lot of enemy units that cannot attack the own units is good. If $uu_{att}(e_j)$ is the number of units in A_{attack} that have no enemy unit in shooting range in the current game state and $uu_{test}(e_j)$ the number of units in A_{tested} with this property, then we have

$$UU(e_j) = uu_{att}(e_j) * w_{uu,att} + uu_{test}(e_j) * w_{uu,test}$$

Again, $w_{uu,att}$ and $w_{uu,test}$ are parameters that determine the influence of this criterion on the fitness.

Danger: While having units that are not engaging the enemy often is bad, it also often is bad to have units that are in range of too many enemy units, allowing for these enemies to collectively attack (and most probably destroy) the unit. The danger criterion Da evaluates this.

$$Da(e_j) = \sum_{i=1}^{u} da(\mathcal{A}g_{attack,i}, e_j) * w_{da,att} + \sum_{i=1}^{n} da(\mathcal{A}g_{tested,i}, e_j) * w_{da,test}$$

Here, $da(\mathcal{A}g, e_j)$ denotes the number of enemy units that could attack agent $\mathcal{A}g$ in the game state e_j and, as usual, $w_{da,att}$ and $w_{da,test}$ are weighting parameters. Since having a lot of units in range of many enemies is a bad thing, $w_{da,att}$ should have a negative value and $w_{da,test}$ a positive one.

Proximity: With only the four criteria above, our test system came often up with an unwanted behavior of its own, namely simply avoiding all enemy units. Therefore we also include as fifth criterion the distance, respectively proximity, of an unengaged unit to the nearest opponent.

$$Pr(e_j) = \sum_{i=1}^{u} pr(\mathcal{A}g_{attack,i}, e_j) * w_{pr,att}$$

Here, pr is the distance between a unit to the nearest enemy unit, if this distance is larger than the attack range and 0, else. And $w_{pr,att}$ is a weight parameter, again.

We bring these criteria together[3] by

$$near_goal((e_0, ..., e_j)) = SU(e_j) + He(e_j) + UU(e_j) + Da(e_j) + Pr(e_j).$$

The genetic operators we use are slight variations of the ones presented in [5]. For two individuals

[3]In fact, in order to fit this value into our general fitness scheme, we would need to divide 1 by it and add l. But since we will stop our search the moment an individual fulfilling $\mathcal{G}_{test}$ is found, our implementation simply sums up the given $near_goal$ over all states and tries to maximize this sum.

$((a_{1,1},...,a_{1,l}),...,(a_{n,1},...,a_{n,l}))$, $((b_{1,1},...,b_{1,l}),...,(b_{n,1},...,b_{n,l}))$ a standard crossover randomly selects a round r and a group of s agents $R = \{\mathcal{A}g^1_{attack}, ..., \mathcal{A}g^s_{attack}\}$ and creates $((c_{1,1},...,c_{1,l}),...,(c_{n,1},...,c_{n,l}))$. We have $(c_{i,1},...,c_{i,l}) = (a_{i,1},...,a_{i,l})$ if $\mathcal{A}g_{attack,i} \notin R$ and $(c_{i,1},...,c_{i,l}) = (a_{i,1},...,a_{i,r-1},b_{i,r},...,b_{n,l})$, else. A standard mutation also randomly selects a round r and an R and creates $((c_{1,1},...,c_{1,l}),...,(c_{n,1},...,c_{n,l}))$. Again, $(c_{i,1},...,c_{i,l}) = (a_{i,1},...,a_{i,l})$ if $\mathcal{A}g_{attack,i} \notin R$ and now $(c_{i,1},...,c_{i,l}) = (a_{i,1},...,a_{i,r-1},a'_{i,r},a_{i,r+1},...,a_{i,l})$ with $a'_{i,r} \neq a_{i,r}$, else.

The targeted variants of these operators select the round r such that it is randomly chosen out of the 5 positions before the evaluation run of the first individual shows a substantial drop in the $near_goal$-value of the environmental state.

2) Testing ORTS with integrated macros: While the basic variant for testing ORTS players from the previous subsection is always able to find individuals that produce behaviors in which two or three attack agents surround a tested agent and eliminate it, winning a game requires a higher level of cooperation between the units of a player, which brings the basic variant to its limits. The known computer players for ORTS employ elaborate tactics (see, for example, [8]) and our testing system should be able to also employ such tactics, which we realize as the cooperation macros we presented in Section III.

We integrated 6 macros that either are aimed at creating formations of the units or to direct all attack agents to attack a particular unit for some time. While the macros themselves do not have parameters (and therefore fit into the evolutionary learning approach we presented), the expansion of the macros performed by an agent's decision function is able to perform computations that allow the use of parameterised actions that the server understands. Our macros use the concept of the center position $center(e_j)$ of all $\mathcal{A}g_{attack,i}$ that are still alive in e_j and that is defined as the average x- and y-coordinates of all these agents (similarly, $oppcenter(e_j)$ is the center position of the $\mathcal{A}g_{tested,i}$). Then the macro actions are as follows:

Assemble: This macro aims at bringing all units together, as the name suggests.$app(Assemble, e_j, k, i)$ is the move action that brings $\mathcal{A}g_{attack,i}$ to $center(e_j)$.

LineUp: This macro aims at having all units forming a vertical line with the center of this line being $center(e_j)$ if the macro is started in e_j. At the start of the macro, the still active units are assigned the position they are supposed to take based on where they start off (by assigning to each unit a number based first on their vertical position and using horizontal position and agent number as tiebreakers and using a given distance between neighbors $dis2Nei$). $app(LineUp, e_j, k, i)$ is then the move action that brings the unit to this assigned position (which can be no move if the unit is already there).

SplitUp: Another often used formation is to split the units into two groups that are equally strong. This macro achieves this by forming one group that positions itself $splitDis$ fields above the $center(e_j)$ position when the macro is activated and one group that positions itself $splitDis$ fields below this position. $splitDis$ is, like $dis2Nei$, a parameter of the testing system. Like for *LineUp*, each agent computes the position it is supposed to go to based on a number assigned to it at the beginning of the macro and then $app(SplitUp, e_j, k, i)$ either is the move action bringing it there or no move, if it is already there.

AttackWeakest: This macro concentrates coordinated attacks on the weakest enemy unit. If $\mathcal{A}g_{attack,i}$ is in range of the enemy unit with the least remaining hit points $app(AttackWeakest, e_j, k, i)$ produces as action shooting at this enemy unit. Else, $app(AttackWeakest, e_j, k, i)$ will result in the move action that brings $\mathcal{A}g_{attack,i}$ nearer to this particular enemy unit. The enemy unit will always be determined based on the current game state e_j.

AttackClosest: This macro is analogous to *AttackWeakest*, except that the target of the cooperative attack is the enemy unit that is closest to $center(e_j)$ (with appropriate tiebreakers common to all agents).

AttackFurthest: This macro is also analogous to *AttackWeakest*, but it targets the enemy unit that is farthest away from $oppcenter(e_j)$.

In our experiments, we had k_{max} being 20 rounds for each of the actions. Since this means that a macro takes up quite a number of rounds (compared to the other actions that need one round), we had the probability for a macro action being produced when creating the random individuals for the first generation of the evolutionary learner at just one percent. The same probability was also used in the mutation operators.

V. Evaluation of our approach

To evaluate the usefulness of cooperation macros, we compare the results of our system for testing ORTS players with and without the use of macros for 3 systems that participated in the 2007 ORTS competition (see [9]). These players are Blekinge, WarsawA and WarsawB and they placed in the middle of the competition (6th, 4th and 3rd, respectively). The rationale behind selecting those players was rather straightforward: we started out trying players that placed in midfield because we assumed that there would be some weaknesses in these players and we could find weaknesses for the three selected players (using macros) within reasonable runtimes[4]. As already stated in Section IV-A, we are not using exactly the competition scenario, which also leads to potential weaknesses, since the players were for sure never tested for the scenario variant we are using.

We used for all our experiments the following parameter settings. A generation of the evolutionary learner consists of 60 individuals (with the 15 best individuals surviving). With an evaluation being playing a game with 10 units on each side for up to 200 rounds, this means that creating and evaluating a generation took on our machines around half an hour (which means that a single run of our system

[4]We also found a weakness for the 5th placed team but only after more than 50 generations, so that including it into the systematic evaluation done in this section would have resulted in much more computing time than the 16 days we needed for all entries about a player.

TABLE I

AVERAGE GENERATION WITH SUCCESS FOR TESTING WITHOUT AND WITH MACROS (SUCCESS RATE IN PARENTHESIS)

Player	Seed	Without	With
Blekinge	1	— (0/4)	15.0 (4/4)
	2	26.5 (2/4)	16.6 (4/4)
WarsawA	1	— (0/4)	3.4 (4/4)
	2	30.0 (1/4)	9.2 (4/4)
WarsawB	1	27.0 (2/4)	9.2 (4/4)
	2	31.3 (4/4)	7.2 (4/4)

takes around a day). For the macro actions aimed at creating formations, we used $dis2Nei = 2$ and $splitDis = 10$. For the fitness evaluation, the weights were set to $w_{uc,att} = 10$, $w_{uc,test} - 10$, $w_{he,att} = 10$, $w_{he,test} = -10$, $w_{uu,att} = 3$, $w_{uu,test} = -3$, $w_{da,att} = 4$, $w_{da,test} = -4$ and $w_{pr,att} = -6$.

As already stated, the ORTS server has the possibility for random effects, like, for example, where the different units of a game start. We "switched off" this feature for a learning run by starting each game we use to evaluate individuals with the same seed for the random number generator. This makes sure that the fitness of an individual is really measuring the behavior for the same problem as the fitness of another individual. The experiments in Table I evaluate the two testing variants (without and with the use of macros) for two different random number seeds (named 1 and 2) and present the average results over 4 runs of the testing system.

As Table I shows, there is quite a difference in the number of generations needed to find the first individual that fulfills $\mathcal{G}_{test}$ with the variant using macros clearly outperforming the variant without macros. The table also shows that sometimes the initial positions for the agents have some influence on the behavior of the agents and consequently on how difficult it is to find a weakness (see WarsawA).

If we look at the computer player weaknesses, then our experiments with macros revealed that Blekinge uses too many rounds to attack. When attacking, all the units stop moving, so that units attacked by the opponent are not running away and the team is also not trying to lure its opponent's units into bad positions. It turned out to be very effective against this player to collectively attack single units (which is easily possible due to the attack macros), eliminating them one by one.

WarsawA splits its units up into two groups one of which seems to try to get behind the opponent team. But it takes a lot of time to do that and therefore starts its attack very late. The teams evolved by our testing system are always able to concentrate on one of the two groups first, eliminate it and then move on to the second group. When dealing with a group, again the attack macros are often used to concentrate on single units and eliminate them quickly.

For WarsawB, our testing system found successful attacks that were in the end similar to what beat WarsawA, but it needed to first entice the player into splitting its team. For this it first used the LineUp macro (see Figure 2, please note that we had to make some colour changes to the usual ORTS screen, since otherwise the attack agents would not have shown in black-and-white printouts), then the four lower attack agents (in the figures, the attack agents are the darker, red agents and the agents/units of the tested player are the light, green ones) drew four player agents down (it seems that the player likes to align its own agents vertically with the opponent's agents) and then come back up again. Figure 3 shows the round when the player's agent groups have nearly come together again, but the larger group has already lost quite some units and the red agents keep pounding on them (using the attack macros). When the upper group is eliminated, the attackers concentrate on the lower group and essentially just before the next to last player agent will be destroyed (see the very small remaining health bar) the 200 rounds are over (see Figure 4).

Due to lack of space, we cannot show any screenshots for successful runs of our testing system without macros. It turns out that analyzing the found behaviors is much more difficult (meaning that it is also more difficult to describe the weakness in the player strategy). The successful runs usually had in the final round only one more surviving agent on the attack agents side than on the tested agents (player) side (so they just barely qualified as a win) and the behaviors of the attack agents were not very coordinated. While we were mainly interested in speeding up the testing process, it turns out that the macros also make it easier to understand the produced results.

VI. RELATED WORK

There are three research areas that we see our work related to: testing of systems, learning of behavior, and automatically creating non-player characters in games. The use of evolutionary search methods for testing of software systems as a sub-area of search-based software engineering is a hot topic for some time now (see [10]), but its use for finding unwanted behavior has not been explored much. In addition to the work in [5], [6], and [7], there is only one other work, namely [11], that can be seen as going into this direction. But [11] does not really test a real system, it tries to find tasks that produce infeasible schedules in the model of a scheduler.

Learning of behavior in multi-agent systems is an active research area for more than 15 years with many very different contributions. [5], for example, was motivated by [12]. But the known approaches in this area focus on achieving a positive goal and enabling agents and teams of agents to achieve this goal under various circumstances, thus learning a set of behaviors or complete agents (see [13] for an overview). Due to this focus, the used agent architectures are much more complex than the action sequences we use.

Automatically creating non-player characters is essentially a subarea of learning of behavior and it also focuses on learning all behaviors for a particular non-player character, which is a much more difficult goal than our goal of finding *one* particular behavior that reveals a problem in a system. Thus, often only certain aspects of the agent are learned (see, for example, the already cited [8] or [14]). Wanting to find only one particular behavior enables us to use agents,

respectively agent architectures, that are less complex than the characters we are testing, resulting in more manageable search spaces and thus in success, as the experiments in the last section have shown.

VII. Conclusion and Future Work

We presented an extension of testing by learning of cooperative behavior that allows the use of so-called coordination macros that represent knowledge about the application a tested system is about. A macro is interpreted by each agent based on the current environmental state and the actions the other agents are taking. Our experimental evaluation showed that with the use of macros our testing system was able to find problematic behavior of computer players for the ORTS environment much faster than without macros and the macros also enabled us to better describe what triggered the problematic behavior, which should make it easier for the developer of a tested system to improve it.

Since macros are about application knowledge, obviously future work should look into other types of applications. Also, despite using macros, the run times for our testing system are rather long, so that other ways of speeding up the search, like distribution or parallelization, should also be employed.

References

[1] J.C. Mogul: Emergent (Mis)behavior vs. Complex Software Systems, Internal Report HPL-2006-2, HP Laboratories Palo Alto, 2005.

[2] Cheat Code Central, http://www.cheatcc.com/, as seen on Mar. 19, 2009.

[3] Trading Agent Competition, http://tac.eecs.umich.edu/, as seen on Feb. 5, 2009.

[4] ORTS - A Free Software RTS Game Engine, http://www.cs.ualberta.ca/∼mburo/orts/, as seen on Feb. 5, 2009.

[5] B. Chan, J. Denzinger, D. Gates, K. Loose, and J. Buchanan: Evolutionary behavior testing of commercial computer games, Proc. CEC 2004, Portland, 2004, pp. 125–132.

[6] J. Denzinger, K. Loose, D. Gates and J. Buchanan: Dealing with parameterized actions in behavior testing of commercial computer games, Proc. CIG-05, Colchester, 2005, pp. 51–58.

[7] J. Denzinger and J. Kidney: Testing the limits of emergent behavior in MAS using learning of cooperative behavior, Proc. ECAI-06, Riva del Garda, 2006, pp. 260–264.

[8] M. van der Heijden, S. Bakkes, and P. Spronck: Dynamic Formations in Real-Time Strategy Games, Proc. CIG-08, Perth, 2008, pp. 47–54.

[9] 2007 ORTS RTS Game AI Competition, http://www.cs.ualberta.ca/∼mburo/orts/AIIDE07/, as seen on Feb. 5, 2009.

[10] P. McMinn: Search-based software test data generation: a survey, Software Testing, Verification and Reliability 14, 2004, pp. 105-156.

[11] L. Briand, Y. Labiche and M. Shousha: Using Genetic Algorithms for Early Schedulability Analysis and Stress Testing in Real-Time Systems, Genetic Programming and Evolvable Machines 7(2), 2006, pp. 145–170.

[12] J. Denzinger and M. Fuchs: Experiments in Learning Prototypical Situations for Variants of the Pursuit Game, Proc. ICMAS-96, Kyoto, 1996, pp. 48–55.

[13] L. Panait and S. Luke: Cooperative Multi-Agent Learning: The State of the Art, JAAMAS 11(3), 2005, pp. 387–434.

[14] C. Miles and S.J. Louis: Case-Injection Improves Response Time for a Real-Time Strategy Game, Proc. CIG-05, Colchester, 2005, pp. 149–156.

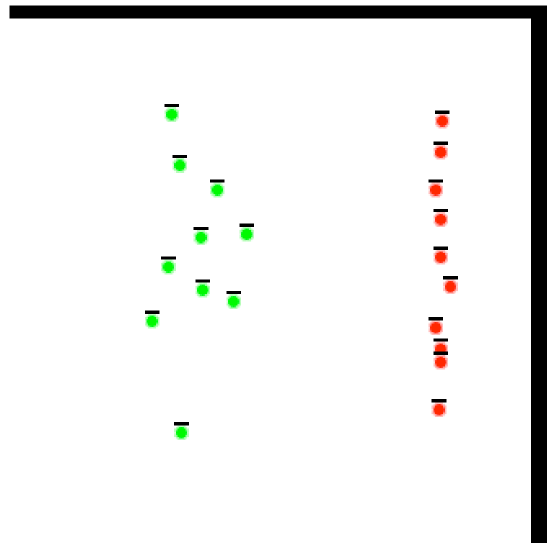

Fig. 2. After LineUp

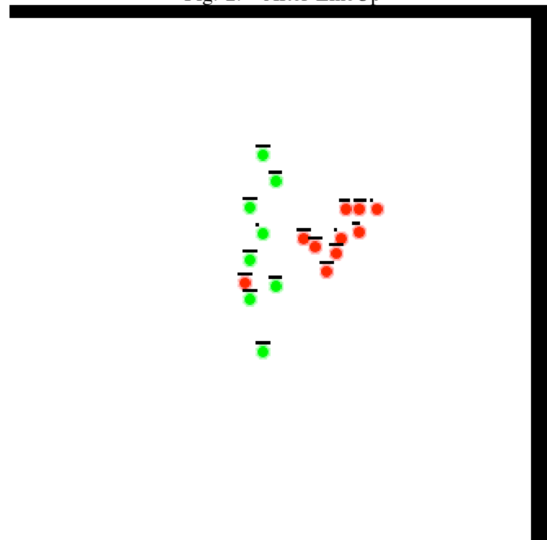

Fig. 3. Enemy too stretched out

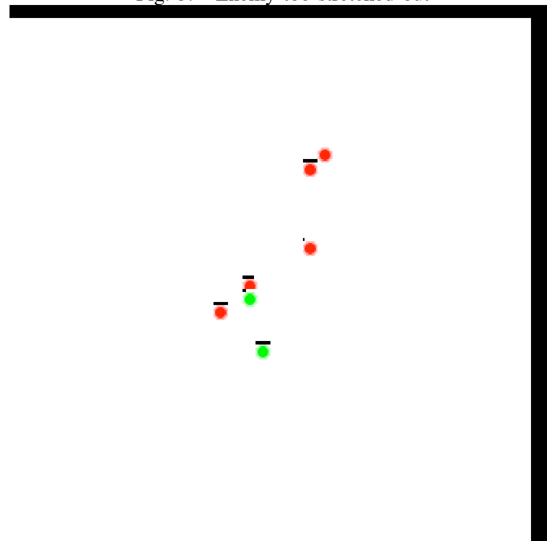

Fig. 4. Game over